Next Up Previous Contents References

3.3 Paths

This section describes how paths are realized in Scout. Scout paths closely follow the model presented in Chapter 2. Since paths make frequent use of lists of name/value pairs, this section begins with a description of *attributes*---the Scout representation of name/value pairs.

Each path is represented by several nested C objects. Since they recursively depend on each other, they cannot be described in a linear fashion without forward references. Thus, the approach taken here is to first present an overview of the objects involved, then to discuss each of them individually in more detail. The descriptions of the objects is followed by an introduction to path dynamics, which involves issues such as how paths are created, used, and destroyed. The section concludes with a brief evaluation and discussion of paths as implemented in Scout.

3.3.1 Attributes

In Scout, an attribute is a name/value pair. The corresponding C type is called `Attr`. The name of an attribute is a (unique) integer index that is usually written in the form of a manifest constant. In contrast, the value can be of arbitrary type. The name of the attribute implies the type of the value. In this sense, an attribute can be thought of as a tagged union variable. Since the association between attribute name and the attribute's value type is by convention, it is not explicitly represented in the implementation. For example, the attribute with name MAX_SIZE might (by convention) have a value of type integer. Similarly, the attribute with name PEER_ADDR might have a value that is a pointer to a network address.

Sets of attributes are implemented in Scout by an abstract type called `Attrs`. A variable of this type can hold an arbitrary number of attributes. The operations supported on attribute sets include insertion, and removal of attributes, enumeration of the attributes present in the set, and membership test.

Why are attributes so important in Scout? They basically serve two different, but equally fundamental roles. During path creation, attributes convey path invariants. In this case, they are primarily a substitute for the variable length argument list support that is missing in the C language (varargs are not defined by the *language*, but by the *runtime system*). But it is more than that. Attribute lists turn argument lists into first **class** objects, that is, they make it possible to enumerate all elements in an argument list under program control, and to dynamically add and remove elements in it.

The second role of attributes is in serving as information repositories that allow mutually anonymous parties to communicate through a third, common party. Consider a path that requires realtime scheduling. For such a path, there typically exists a deadline by which a task has to be completed. The entity that can compute the deadline may be somewhere in the middle of the path, but a device driver that enqueues newly arrived data for the path may need to know what the current deadline is, so it can schedule the path in an appropriate manner. This problem can be solved easily by using an attribute stored in the path. All that is required is that the users of attributes agree that, for example,

attribute DEADLINE specifies a task deadline in units of micro-seconds. This attribute attached to the path allows communicating the deadline from the middle of the path to the device driver at the edge of the path. Note that this communication is anonymous: the producer of the deadline does not know who makes use of the attribute and conversely, the consumer does not know who produced the attribute. Also, the **path object** holding the attributes does not have to understand the meaning of the attributes stored in it.

3.3.2 Visual Overview of a Scout Path

Figure 14 presents an enlarged view of the path and module graph originally presented in Figure 5. The path is shown as the big rectangular box in the right half of the figure. For space reasons, the middle of the path has been cut out of the figure. Internally, the path contains four queues, shown near the top and bottom of the box representing the path. It also contains several smaller boxes called *stages*. As indicated, there is one stage per module that the path traverses. Inside stages there are various labelled arrows. The exact meaning of these arrows will be described in Section 3.3.4. Aside from the queues and the stages, the path contains other minor objects which have been omitted in the figure for the sake of clarity.

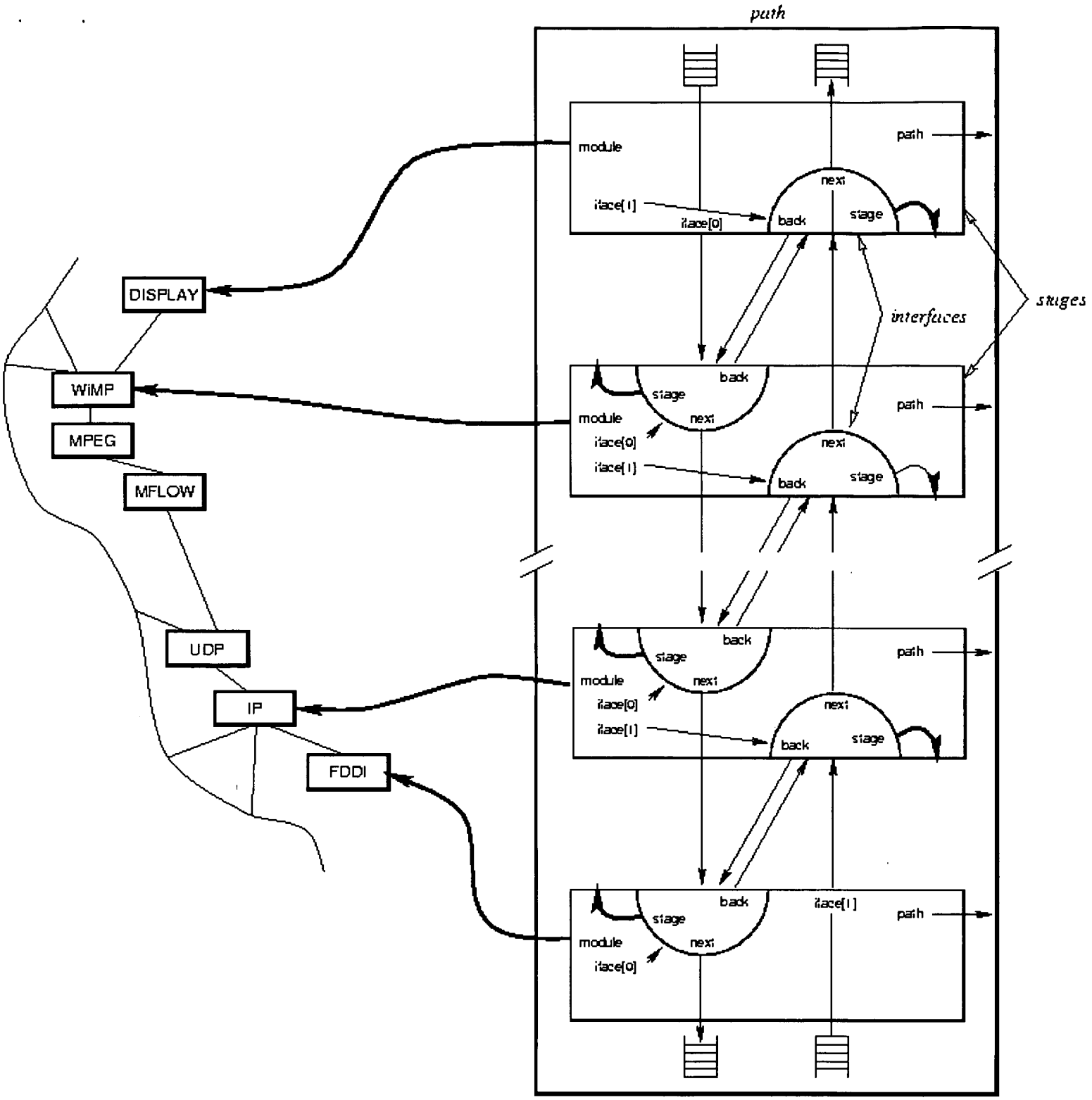

Figure 14: Path Structure

The stages inside the path are created by the modules along the path; e.g., the figure suggests that the bottom-most stage was created by FDDI. Stages provide a place to store information that is path-specific, but private to the modules. For example, the stage created by module IP might contain a pre-formatted IP-header that is used when sending data from the path to the network. Since no other module along the path needs to know about IP headers, it is best to store it in the stage created by module IP.

The figure also shows that stages contain semi-circular shapes, called *interfaces*. An interface provides a controlled

(type-checked) way to move data from one stage to the next one. Interior stages typically have a pair of interfaces (one per direction) whereas stages at the ends of the path typically contain just one interface.

The figure further shows that interfaces are chained together. The interfaces in the left half of the path are used to move data from the top end of the path towards the bottom, whereas the interfaces in the right half are used to move data from the bottom end towards the top. As discussed in Chapter 2, it is sometimes necessary to turn around the direction in which data flows. This is supported by the arrows in the interfaces that are labelled *back*. That is, when data arrives at an interface, it is possible to continue moving it in the same direction by **passing** it along the arrow labelled *next*, or it can be turned around and passed along the arrow labelled *back*.

3.3.3 Path Object

The most important elements of the actual Scout **path object** are shown in the C structure below. In addition to the members shown, the Scout **path object** contains state that assists the creation, extension and destruction of paths. That state has been omitted since it is not relevant to the discussions presented here.

```
typedef struct Path {
  long          pid;
  Stage         end[2];
  PathQueue     q[4];
  struct Attrs  attrs;
  bool          realtime;
  u_long        prio;
} * Path;
```

Every path has a unique integer associated that is called the *path id*. This id is stored in member `pid`, and permits accounting resources on a per-path basis. The id is guaranteed to be unique, but is otherwise arbitrary. In particular, if n paths exist in the system, it is *not* guaranteed that the path id is in the range 0..n-1.

The stages at the extreme ends of the path are pointed to by `end[0]` and `end[1]`. For an unextended path, `end[0]` refers to the stage created first, whereas `end[1]` refers to the stage created last. If a path is extended at the end pointed to by `end[0]`, then, once path extension is complete, `end[0]` will point to the last stage created during extension. The corresponding applies for path extension applied to `end[1]`. In other words, `end[0]` and `end[1]` are guaranteed to point to the stages at the opposing ends of the path---which expression points to which end is, in general, difficult to say. As explained in Section 2.2.4.1, this is due to the fact that paths are completely symmetric as far as the two data-transmission directions are concerned.

The four path queues can be accessed through array `q`. The path queues are implemented in the stages at the ends of the path, so array `q` consists of pointers to the actual queues. If a stage does not implement a particular **queue**, then the corresponding pointer in `q` is `NULL`. The mapping between the **queue** index and the function of the **queue** is given in the following table:

expression	functions as...	in direction...
`q[0]`	source	forward (`end[0]->end[1]`)
`q[1]`	sink	forward (`end[0]->end[1]`)
`q[2]`	source	backward (`end[1]->end[0]`)
`q[3]`	sink	backward (`end[1]->end[0]`)

To leave the stages flexibility in choosing the implementation for each path **queue**, only two **queue** operations are globally defined: one to determine the maximum length of the **queue** and another to determine the current length of the **queue**. These operations tend to be sufficient to assist in making resource management decisions since they allow computing the percentage to which a **queue** is full. Other operations that could prove useful in the future are certainly imaginable, however. One example is an operation that would allow resizing the path **queue**.

The attribute set `attrs` provides a place to associate arbitrary information with the path. In essence, it provides the means to dynamically (at runtime) expand the **path object**. This is useful to communicate information between different

stages in the path and also encourage exploring new, path-related ideas. The latter is true since the attribute set makes it possible to associate new state with a path without requiring compile-time modifications the **path object** type. Also, path attributes are useful for performance measurements. It is often the case that the measurements are performed in one part of the path, but reported in another, or that measurements need to be accumulated over a certain period of time. Both cases can be accommodated easily using attributes to store performance statistics in the **path object**.

The last two members listed in the structure are `realtime` and `prio`. These represent the scheduling parameters of the path and will be described in more detail later in Section 3.5.

3.3.4 Stage

Figure 14 shows that path-interior stages essentially represent fixed routing decisions: data enters the stage, which is then processed by code specific to the module that created the stage, and, eventually, leaves the stage through the other side. In a traditional system without paths, the module-specific processing would also require a routing decision to determine where to send the data next. In contrast, interior stages have these decisions pre-made (builtin). However, the stages at the ends of the path are special. They do not represent fixed routing decisions. Instead they simply serve as hooks into the modules at which the path terminates. These are the modules that will need to make dynamic routing decisions to find out where to send the data next. In the ideal case, a path terminates at modules at the edge of the module graph, where data is simply passed on to the appropriate device. For this case, all dynamic routing decisions can be avoided.

The C structure that implements the stage object in Scout is shown below:

```
typedef struct Stage {
  Iface  iface[2];
  Path   path;
  Module module;
  long   (*establish)(Stage s, Attrs a);
  void   (*destroy)(Stage s);
  Stage  nextStage;
} * Stage;
```

The elements of array `iface` are pointers to the interfaces in the stage. If such an interface does not exist, the corresponding element is `NULL`. Expression `iface[0]` refers to the interface on the side of the stage through which the creation request for this path arrived (see Section 3.3.6). For an unextended path, this is equivalent to the interface in the forward direction of the path. Expression `iface[1]` refers to the other interface in the stage. For an unextended path, this is equivalent to the interface in the backward direction of the path. For stages that were created as part of a path extension, this simple and direct relationship between the elements in array `iface` and the path direction does not hold, however.

The path to which the stage belongs is pointed to by `path`. Similarly, the module that created the stage can be accessed through member `module`. The function pointers `establish` and `destroy` are used during path creation and destruction and are explained in detail later in this section. The final member, `nextStage`, is part of a chain that lists all of a path's stages in the order in which they were created. Due to path extension, this chain does not necessarily correspond to a linear traversal of the path and hence is typically used during path creation and destruction only.

3.3.5 Interface

As mentioned in the overview, interfaces are used to move data from one stage to the next one. The simplest possible interface has the structure shown below:

```
typedef struct Iface {
  Iface next;
  Iface back;
  Stage stage;
```

```
} * Iface;
```

In words, the most primitive interface simply consists of a `next` pointer that refers to the next interface in the current direction of the path, a `back` pointer that refers to the next interface in the opposite direction, and a `stage` pointer that refers to the stage that the interface belongs to. This is simple, but does not provide any way to deliver data to the interface. Thus, all useful interfaces are expanded versions of this primitive interface. That is, Scout uses single inheritance for interfaces [38]. For example, the interface that is commonly used to pass a ***message*** (a sequence of data bytes) to an interface is called `AioIface` (asynchronous I/O interface). This interface looks the same as the basic `Iface`, except that it additionally declares a function pointer that can be used to deliver a **message**:

```
typedef struct AioIface {
  struct Iface i;
  long         (*deliver)(Iface i, Msg m);
} * AioIface;
```

To deliver **message** `m` to the asynchronous I/O interface `i`, one would simply invoke `i->deliver (i, m)`.

Since interfaces provide for single inheritance, the service compatibility rules presented in Section 3.2.3.1 can be slightly relaxed: if a service requires an interface of type T, then any interface type that is equal to or a subtype of T can be used to satisfy this requirement. For example, if interface type `Iface` were required, then an interface of type `AioIface` could be used instead. This is sensible since any asynchronous I/O interface is also a plain interface. A more interesting example involves TCP since, to a first approximation, TCP provides a bytestream, it clearly would be desirable if it were possible to connect TCP to any other module that requires a bytestream. However, a sophisticated user of TCP might want to adjust its flow-control window sizes, so it would be useful if TCP provided functions to do so. With single inheritance, TCP can use a subtype of `AioIface` that provides the additional functions needed to adjust the window sizes. Suppose that subtype was called `WindowedAioIface`. With this arrangement, a sophisticated user that depends on being able to control window sizes could specify a service that requires an interface of type `WindowedAioIface`, whereas naive users could continue to connect to TCP as if it were a regular asynchronous I/O interface.

Technically, Scout can support an arbitrary number of interface types. However, the more interface types there are, the less likely that any given pair of services can be connected. Thus, the intent is to keep this number as small as possible. At present, there are about eight different interface types that are relatively stable and in frequent use. In addition to those, there are another eight interface types related to the disk subsystem. Since that subsystem is still being evolved, its interfaces are not as well factored and stable as the others. The simplest useful interface, asynchronous I/O, is employed by many data filters and almost all modules related to networking. The most complicated interface so far is a window management system interface that defines more than thirty distinct operations.

The decision as to whether a new module should be coerced into using existing interface types or whether a new interface type should be defined instead is sometimes not easy. It should be guided by the semantics of the operation involved and by performance considerations. Semantics must be taken into consideration to ensure that compatible services really result in meaningful behavior when connected. If performance were not taken into account, there would be no reason to support multiple interfaces, since everything could be built based on a universal data-delivery function such as the `deliver` function in the asynchronous I/O interface.

3.3.6 Creation

Now that all parts of the Scout path have been described, it is possible to explain the details of path creation. Scout provides a single `pathCreate` function for this purpose. Its C prototype is given by:

```
Path pathCreate (Module m, Attrs a);
```

As the prototype suggests, a path is created by invoking the function on a module `m` with an attribute set `a`. The attribute set describes the kind of path that is desired. That is, the invariants discussed in Chapter 2 are passed in this set.

The call to `pathCreate` eventually results in an invocation of the `createStage` function in module *m* (see Section 3.2.3.2). The `createStage` function has the following type:

```
Stage (*CreateStageFunc)(Module m, int s, Attrs a,
                         ModuleLink* n);
```

In the prototype, argument `m` is the module on which `pathCreate` was invoked and `s` is the index of the service through which the path being created entered the module. Since `m` is the first module in the path, there is no such service, so a value of -1 is passed (which is not a valid service index). Argument `a` is the set of attributes that was passed to the `pathCreate` function. In response to this invocation, the `createStage` function is expected to allocate and initialize a stage and the interfaces contained therein. As part of this processing, the function may also update the attribute set as new information about the path may become available in the module or existing information may become obsolete. After creating the new stage, the module attempts to make a routing decision based on the attributes (invariants) that were passed to it. If the module can decide where the path has to go next, it sets `*n` to point to the module and service index of that next point. If no routing decision can be made based on the attributes, then path creation stops at this module and `*n` is set to `NULL`.

If the call to `createStage` returned a non-`NULL` value in `*n`, then path creation continues at the point given by `*n`. This is done by invoking `createStage` on the module indicated by `*n` and with argument `s` set to the service index specified in `*n`. The attribute set `a` is the possibly updated set returned by the previous `createStage` invocation. The reason the service index `s` is passed to the stage creation routine is because stages usually need to be created differently depending on the service through which the path entered the module. In a sense, the service index is a very short-lived path invariant, but since it changes so frequently (with every stage creation call), it is more efficient and more convenient to pass it as a separate function argument. Given the service index and the current attribute set, a new stage is created and a routing decision made and stored in `*n`, if possible. This process repeats until the path reaches its full length which happens either when it reaches a leaf module or when the attributes are too weak for a module to make a static routing decision.

Once the path has reached its full length, a sequence of stages exists. At this point, the `pathCreate` function creates the actual **path object**, inserts the stages into it, and establishes the various chains through the path structure. In a third step, the `establish` callbacks in the stage objects are invoked in the order in which the stages were created. The establish callbacks are necessary since some stages cannot be fully initialized until the entire path structure exists. The first argument passed to the `establish` callback is a self-reference to the stage being established and the second argument is an attribute set. This attribute set is initialized to the empty set before invoking the first callback. The establish callbacks may use and modify this attribute set as necessary and then pass it on to the next establish callback. The purpose of this attribute set is to allow **passing** auxiliary information between neighboring stages. This can be done safely since neighboring stages are known to be compatible in the sense defined in Section 3.2.3.1. An example for such auxiliary information is attribute `PREVIOUS_DEMUX_NODE`, which will be described in Section 3.4.3. Note that the attribute set used in the establish callbacks has nothing in common with path invariants or the attribute set passed to `pathCreate`.

3.3.7 Extension

The path extension function operates analogously to the path creation function. The only difference is that path extension is invoked at the end of an existing path, rather than on a module. The prototype for this function is shown below:

```
long pathExtend (Stage s, Attrs a);
```

The first argument, stage `s`, points to the end of the path that should be extended. If path *p* is being extended, this must be either *p*`->end[0]` or *p*`->end[1]`. If `s` points to any other stage, path extension will fail. The second argument, attribute set `a`, is the set of invariants that are true for the path extension operation.

If path extension fails for any reason (e.g., because the system runs out of memory), then the path being extended is

destroyed as well. This fate sharing is reasonable since an extended path is still just one path; it does not consist of two independent sub-paths. It is therefore only logical that if path extension fails, then the entire path creation should be considered to have failed.

For path extension to make sense, the attribute set must be consistent with the one specified in the `pathCreate` call and those specified during previous calls to `pathExtend` (if any). Scout does not have a formal model for path invariants. Thus, it is not possible to give a formal procedure to test whether a pair of attribute sets is consistent. Informally, it is easier to discuss cases that make attribute set pairs inconsistent. For example, if the first set contains the invariant that the path needs to be scheduled using a realtime scheduler and the second set contains an invariant that request a best-effort scheduler, then the attribute sets are inconsistent. Such direct contradictions are relatively easy to detect. More subtle are inconsistencies that arise from *not* specifying invariants; e.g., the first attribute set may contain an invariant that says no single data-item (**message**) is going to be larger than 100 bytes, while the second attribute set may not have any invariants related to the size of messages, so even though there is no direct contradiction in the invariants, the attribute sets may be inconsistent.

The reason we introduced the issue of attribute set consistency here is that the problem is most apparent with path extension. However, it is not limited to this operator. Since the create-stage functions may update the attribute set, consistency problems may arise within a single path creation operation. Note that prohibiting stage creation from updating the invariant set is not a solution since modules do process and modify data, and as a result, invariants may have to be updated to reflect those changes.

3.3.8 Optimization

The current incarnation of Scout does not employ an explicit `pathOptimize` function to apply the path transformations described in Chapter 2. Instead, path transformations are applied in an ad hoc fashion. This certainly will not be sufficient in the long run, but does have the advantage of providing maximum flexibility in experimenting with path transformations. A sample path transformation designed to improve processing speed inside will be discussed in detail in Chapter 4.

While no explicit path optimization routine exists, it is important to point out that it is straight-forward to replace the code of a path with a more optimized version. This is because interfaces contain function pointers, not actual code. Hence, to replace the code used by a path, all that needs to be done is change the function pointers in the interfaces to make them point to the optimized versions.

3.3.9 Destruction

The final path-related operation allows one to destroy a path when it is no longer needed. The C prototype for this operation is shown below:

```
void pathDelete (Path p);
```

Invocation of this function will eventually cause path `p` to be destroyed. However, before this happens, the `destroy` callbacks in the path's stages are called in the order in which the stages were created. The only argument passed to this callback is the stage (and, implicitly, the path) that is being destroyed. The destroy callback of a stage needs to ensure that all resources held by the stage are relinquished. Once all callbacks have been executed, the resources held by the path are relinquished and the path ceases to exist.

3.3.10 Evaluation and Discussion

As implemented in Scout, paths are light-weight. For example, a path to transmit and receive UDP network packets consists of six stages. Creating such a path on a first-generation, 21064A 300MHz Alpha takes on the order of 200µs (not including the application of any potential path optimization transformations). The **path object** itself is about 300 bytes long and each stage is on the order of 150 bytes in size, including all the interfaces. In total, each UDP path takes

about 1200 bytes. Since interfaces consist of pointers to functions, creating a path does not cause any code duplication. Code duplication can occur only as a result of path transformations.

The current Scout architecture does not support multiple protection domains. However, extending Scout paths to multiple protection domains should be straight-forward. Indeed, paths raise the interesting opportunity to use protection not just between *layers* (horizontal partitioning) but also between *paths* (vertical partitioning). With horizontal partitioning, layers are protected from each other, whereas with vertical partitioning the paths are protected from each other. Depending on the needs of a system (e.g., debugging of a new layer versus ensuring the integrity of the data sent through a path), one or the other or even a combination of the two may be appropriate.

For horizontal partitioning, note that all communication between stages is through a pair of interfaces. Splitting a path at such a boundary into two protection domains would therefore be rather natural and easy. Another concern is that the actual **path object** needs to be accessible from all protection domains that a path crosses. Different solutions are conceivable. One would involve caching the **path object** in the different domains. Another would involve keeping the **path object** in a protection domain that is accessible by all other domains. A third would involve accessing the **path object** through a system call-like interface. Most likely, an actual implementation would use a combination of the three proposed solutions, but the key point is that there do not seem to be any unusual difficulties in defining paths that cross multiple protection domains.

Vertical partitioning does not appear to be problematic either. The entire path would be contained in its own protection domain and the domain would have to be crossed only when moving from a module into a path or vice versa. Ideally, paths directly connect modules representing device pairs, so the number of domain crossings for moving data from a source device to sink device would be two, just as is the case with a traditional, monolithic kernel based system [95].

Next Up Previous Contents References